Alessandro Turrin s286290 Appello del 26/01/2021

STRUTTURE DATI

1. *Graph*: ADT I^ classe, avente come campi:
   1.1. numero di vertici 'V' e archi 'E', di tipo *int*;
   1.2. matrice di adiacenza 'madj', di tipo *int***;
   1.3. tabella di simboli 'tab', anch'essa ADT I^ classe;
   1.4. vettore dinamico 'stanza' di tipo *Stanza*;
   1.5. lista di adiacenza 'ladj', alla fine non effettivamente utilizzata.

2. *Stanza*: quasi ADT, avente come campi:
   2.1. 'nome', di tipo *char**, da allocare;
   2.2. 'profondità', 'tesoro' e 'oro', di tipo *int*.

3. *Path*: quasi ADT, avente come campi:
   3.1. dimensione 'N', di tipo *int*;
   3.2. vettore dinamico 'stanza' di tipo *Stanza*;
   3.3. vettore dinamico 'raccolto' di tipo *int*, per segnalare se l'oro fosse stato raccolto o meno da una stanza.

ALGORITMI E STRATEGIE RISOLUTIVE

DOMANDA 9

La funzione *GRAPHload()* è stata implementata secondo il medesimo modello studiato durante il corso. Viene effettuata innanzitutto la lettura del numero di stanze 'S', chiamando poi la funzione di libreria *GRAPHinit()*, la quale effettua l'allocazione e la prima inizializzazione del grafo 'G'. Successivamente, viene effettuata l'allocazione del vettore di tipo *Stanza*, per poi proseguire con la lettura da file. Infine, viene effettuata la lettura degli archi del grafo, secondo una struttura *<stanza n> <stanza m> <trappola>*. Per quanto riguarda la trappola, se presente, nella matrice di adiacenza del grafo pesato non orientato G viene assegnato il valore 2, mentre in caso contrario il valore assegnato è 1.

DOMANDA 10

a. GRAPHpathLoad()

Come primo passo, si effettua la lettura da file del numero di stanze presenti, per poter allocare i vettori dinamici 'p.stanza' e 'p.raccolta'. In sequenza:

- si effettua l'input dei nomi delle stanze;
- si cerca nella tabella di simboli 'g->tab', mediante la funzione *STsearch()*, il corrispondente indice nel vettore dinamico 'g->stanza';
- si inizializza correttamente il vettore 'p.stanza'. Il campo 'p.stanza[i].nome' viene inizializzato mediante la funzione di libreria *strdup()*.

b. GRAPHpathCheck()

Innanzitutto, si effettua un primo controllo sui punti di inizio e di fine del percorso compiuto che, se si dovessero rivelare non accettabili, permetterebbero un'anticipazione del *return 0*, in modo tale da evitare ulteriori e superflui passaggi. Si entra successivamente nel ciclo *for*, il quale, ad ogni iterazione, verifica se il numero di mosse, contate da 'i', sia superiore al numero di mosse totali concesse 'M' e, in tal caso, la funzione ritorna esito negativo.

Con la funzione *STsearch()* si verifica se le stanze presenti nel file siano effettivamente presenti nel grafo o meno. Successivamente si verifica l'effettiva presenza di un cunicolo tra due stanze, mediante l'utilizzo della matrice di adiacenza del grafo: se il valore corrispondente a 'g->madj[i1][i2]' è maggiore di zero, allora le due stanze sono adiacenti, mentre in caso contrario la funzione ritorna esito negativo.
Nel caso in cui 'g->madj[i1][i2]' sia 2, ossia il valore che identifica la presenza di una trappola, i punti vita 'PF' dell'esploratore vengono decrementati di uno.

DOMANDA 11

a. *GRAPHpathBest()*
   La funzione è stata utilizzata come *wrapper* della funzione ricorsiva *GRAPHpathBestR()*.
   Vengono innanzitutto allocati i vettori dinamici 'p.stanza' e 'pBest.stanza', rappresentanti rispettivamente la soluzione corrente e la soluzione ottima, per poi effettuare la chiamata alla funzione ricorsiva.

b. *GRAPHpathBestR()*
   La funzione ricorsiva, a partire dal vertice di inizio 'v' (con valore iniziale 0, ossia il punto di ingresso del labirinto), visita il grafo secondo l'adiacenza dei vertici, per arrivare ricorsivamente al vertice finale 'w' (anch'esso con valore 0), non avendo erroneamente tenuto in considerazione il fatto che l'esploratore non debba necessariamente uscire dalla grotta, ma che possa essere soccorso fino ad una profondità minore o uguale a 2.
   All'interno del ciclo for, si effettua un controllo iniziale sulla raccolta dell'oro, questione gestita successivamente in fase di *backtrack*. La chiamata ricorsiva viene effettuata per visitare in modo esaustivo il grafo, aggiornando di volta in volta il valore di 'v', ma tenendo fisso il valore di 'w'.
   In fase terminale, viene verificata la validità della soluzione e la raccolta complessiva di risorse. Se la verifica risulta essere migliorativa rispetto alla 'pBest', quest'ultima viene aggiornata con la soluzione corrente 'p'.

ELENCO DIFFERENZE

Struttura dati

1. E' stata rimossa la lista di adiacenza, poiché risultata superflua.
2. Aggiunta dei campi 'oroTotale', 'tesoro' e 'totale' al quasi ADT Path, per poter ridurre significativamente il numero di parametri nella funzione ricorsiva *GRAPHpathBestR()*.

Domanda 9

1. Definita la variabile statica 'nome' di tipo char[100], omessa in fase iniziale, sostituendo nella *fscanf()* del primo ciclo *for* la variabile 'id1' con 'nome'.

Domanda 10

1. *GRAPHpathLoad()*
   1.1. Riscritta correttamente la *calloc()*.
2. *GRAPHpathCheck()*
   2.1. Aggiunta la condizione di controllo preliminare sulla validità della profondità dell'ultimo elemento del percorso.
   2.2. Sostituita la svista 'g->madj[i1][i1]' con la forma corretta 'g->madj[i1][i2]'.

Domanda 11

1. *GRAPHpathBest()*
   1.1. Effettuate le inizializzazioni complete dei campi delle variabili ‘p’ e ‘pBest’.
   1.2. Inizializzata di default ‘p.stanza[0]’ con ‘Ingresso’.
   1.3. Rimosse le variabili superflue poiché aggiunte come campi del quasi ADT Path.
2. GRAPHpathBestR()
   2.1. Sostituita la condizione finale con ‘(pos>=M || (pos>0 && v==0))’.
   2.2. Sostituita la condizione di verifica delle risorse con la funzione di verifica *calcolaRisorse()*, in quanto più efficiente.
   2.3. Aggiunta la funzione *PATHcreate()* per poter copiare la soluzione ‘p’ in ‘pBest’.
   2.4. Sostituita interamente la parte sulla lista di adiacenza con una verifica sulla matrice di adiacenza, al fine di evitare di dover aggiungere una struttura dati all’ADT I^ classe Graph, ritenuta non necessaria.
   2.5. Il tesoro attuale viene sostituito con un tesoro migliore, se rilevato, per poi applicare il *backtrack* nel caso in cui la stanza in esame non venga presa.
   2.6. Migliorato il *backtrack* sulla raccolta dell’oro, considerando ora i passaggi multipli in modo più completo, senza limitarsi al semplice caso ‘preso/non preso’.